Apprentissage Statistique avec Python.scikit-learn

Résumé

Après la préparation des données, cette vignette introduit l'utilisation de la librairie `scikit-learn` *pour la modélisation et l'apprentissage. Pourquoi utiliser* `scikit-learn` *? Ou non ? Liste des fonctionnalités, quelques exemples de mise en œuvre : exploration (ACP, AFCM, k-means), modélisation (régression logistique, k-plus proches voisins, arbres de décision, forêts aléatoires. Optimisation des paramètres (complexité) des modèles par validation croisée.*

- *Python pour Calcul Scientifique – Statistique*
- *Trafic de Données avec Python.*`Pandas`
- *Apprentissage Statistique avec Python.*`Scikit-learn`
- *Sciences des données avec Spark.mllib*

1 Introduction

1.1 `Scikit-learn` *vs.* R

L'objectif de ce tutoriel est d'introduire la librairie `scikit-learn` de Python dont les fonctionnalités sont pour l'essentiel un sous-ensemble de celles proposées par les librairies de R. Se pose alors la question : quand utiliser `scikit-learn` de Python plutôt que par exemple `caret` de R plus complet et plus simple d'emploi ?

Le choix repose sur les points suivants :

- Attention cette librairie manipule des objets de classe `array` de `numpy` *chargés en mémoire* et donc de taille limitée par la RAM de l'ordinateur ; de façon analogue R charge en RAM des objets de type `data.frame`.
- Attention toujours, `scikit-learn` (0.16) ne reconnaît pas (ou pas encore ?) la classe `DataFrame` de `pandas` ; `scikit-learn` utilise la classe `array` de `numpy`. C'est un problème pour la gestion de variables qualitatives complexes. Une variable binaire est simplement remplacée par un codage $(0, 1)$ mais, en présence de plusieurs modalités, traiter celles-ci comme des entiers n'a pas de sens statistique et remplacer une variable qualitative par l'ensemble des indicatrices (*dummy* variables$(0, 1)$) de ses modalités complique les stratégies de sélection de modèle et rend inexploitable l'interprétation statistique.
- Les implémentations en Python de certains algorithmes dans `scikit-learn` sont aussi efficaces (*i.e.* k-means), voire beaucoup plus efficaces (*i.e.* forêts aléatoires) pour des données volumineuses.
- R offre beaucoup plus de possibilités pour une exploration, des recherches et comparaisons de modèles, des interprétations mais les capacités de parallélisation de Python sont plus performantes, de même que l'étape initiale de *data munging* de données volumineuses.

En conséquences :

- Préférer R et ses libraires si la présentation (graphiques) des résultats et leur interprétation est prioritaire, si l'utilisation et / ou la comparaison de beaucoup de méthodes est recherchée.
- Préférer Python et `scikit-learn` pour mettre au point une chaîne de traitements (*pipe line*) opérationnelle de l'extraction à une analyse privilégiant la prévision brute à l'interprétation et pour des données quantitatives ou rendues quantitatives ("vectorisation" de corpus de textes).

En revanche, si les données sont trop volumineuses pour la taille du disque et distribuées sur les nœuds d'un *cluster* sous *Hadoop*, consulter l'étape suivante pour l'utilisation de `MLlib` de Spark/Hadoop.

1.2 Fonctions de `scikit-learn`

La communauté qui développe cette librairie est très active, elle évolue rapidement. Ne pas hésiter à consulter la documentation pour des compléments. Voici une sélection de ses principales fonctionnalités.

- Transformations (standardisation, discrétisation binaire, regroupement de modalités, imputations rudimentaires de données manquantes) , "vectorisation" de corpus de textes (encodage, catalogue, Tf-idf), images.
- Exploration : ACP, classification non supervisée (mélanges gaussiens, propagation d'affinité, ascendante hiérarchique, SOM,...)
- Modéle linéaire général avec pénalisation (ridge, lasso, elastic net...), analyse discriminante linéaire et quadratique, k plus proches voisins, processus gaussiens, classifieur bayésien naïf, arbres de régression et classifica-

tion (CART), agrégation de modèles (bagging, random forest, adaboost, gradient tree boosting), SVM (classification, régression, détection d'atypiques...).
- Algorithmes de validation croisée (loo, k-fold, VC stratifiée...) et sélection de modèles, optimisation sur une grille de paramètres, séparation aléatoire apprentissage et test, enchaînement (pipe line) de traitements, courbe ROC.

En résumé, cette librairie est focalisée sur les aspects "machine" de l'apprentissage de données quantitatives (séries, signaux, images) volumineuses tandis que R intègre l'analyse de variables qualitatives complexes et l'interprétation statistique fine des résultats au détriment parfois de l'efficacité des calculs.

Les différences d'usage entre R et `Pyhton.scikitlearn` résument finalement les nuances qui peuvent être mises en évidences entre apprentissage *Statistique* et apprentissage *Machine*.

1.3 Objectif

L'objectif est d'illustrer la mise en œuvre de quelques fonctionnalités [1] de la librairie `scikit-learn`.

Deux jeux de données élémentaires sont utilisés. Celui précédent géré avec `pandas` et concernant le naufrage du Titanic mélange des variables explicatives qualitatives et quantitatives dans un objet de la classe `DataFrame`. Pour être utilisé dans `scikit-learn` les données doivent être transformées en un objet de classe `Array` de `numpy` en remplaçant les variables qualitatives par les indicatrices de leurs modalités. L'autre ensemble de données est entièrement quantitatif. C'est un problème classique de reconnaissance de caractères qui est inclus dans la librairie `scikit-learn`, de même que les trop fameux iris de Fisher.

Ces données sont explorées par analyse en composantes principales (caractères) ou analyse multiple des correspondances (titanic), classifiées puis modélisées par régression logistique (titanic), k- plus proches voisins (caractères), arbres de discrimination, et forêts aléatoires. Les paramètres de complexité des modèles sont optimisés par minimisation de l'erreur de prévision estimée par validation croisée.

D'autres fonctionnalités sont laissées momentanément de côté ; cela concerne les possibilités d'enchaînement (*pipeline*) de méthodes et d'automatisation. Il s'agit, par exemple, de répéter automatiquement la séparation aléatoire des échantillons apprentissage et test pour estimer les distributions des erreurs, comme c'est relativement facile à mettre en œuvre en R avec la librairie `caret` [2]. Ces fonctionnalités seront développées en Python avec les scénarios à venir sur des données plus complexes et plus volumineuses.

2 Exploration multidimensionnelle

2.1 Les données

Les données "Caractères"

Il s'agit d'explorer celles de reconnaissance de caractères dont les procédés d'obtention et prétraitements sont décrits sur le site de l'UCI (Lichman, 2013)[3]. Les chiffres ont été saisies sur des tablettes à l'intérieur de cadres de résolution 500×500. Des procédures de normalisation, ré-échantillonnage spatial puis de lissage ont été appliquées. Chaque caractère apparaît finalement discrétisé sous la forme d'une matrice 8×8 de pixels à 16 niveaux de gris et identifié par un label. Les données sont archivées sous la forme d'une matrice ou tableau à trois indices. Elles sont également archivées après vectorisation des images sous la forme d'une matrice à $p = 64$ colonnes.

Ouvrir la fenêtre d'un calepin `ipython` dans un navigateur.

```
# Importations
import matplotlib.pyplot as plt
from sklearn import datasets
%pylab inline
# les données
digits = datasets.load_digits()
# Contenu et mode d'obtention
print(digits)
# Dimensions
digits.images.shape
# Sous forme d'un cube d'images 1797 x 8x8
```

1. Consulter la documentation et ses nombreux exemples pour plus de détails.

2. Consulter les scénarios de wikistat.fr.

```
print(digits.images)
# Sous forme d'une matrice 1797 x 64
print(digits.data)
# Label réel de chaque caractère
print(digits.target)
```

Voici un aperçu des images à discriminer :

```
images_and_labels = list(zip(digits.images,
   digits.target))
for index, (image, label) in
   enumerate(images_and_labels[:8]):
     plt.subplot(2, 4, index + 1)
     plt.axis('off')
     plt.imshow(image, cmap=plt.cm.gray_r,
        interpolation='nearest')
     plt.title('Training: %i' % label)
```

Titanic

Les données sur le naufrage du Titanic sont décrites dans le scénario consacré à `pandas`. Reconstruire la table de données en lisant le fichier `.csv` disponible dans ce répertoire ou charger l'archive au format HDF5.

```
# Lire les données d'apprentissage
import pandas as pd
df=pd.read_csv("titanic-train.csv",skiprows=1,
  header=None,usecols=[1,2,4,5,9,11],
  names=["Surv","Classe","Genre","Age",
    "Prix","Port"],dtype={"Surv":object,
    "Classe":object,"Genre":object,"Port":object})
df.head()
# dimensions
df.shape
# Redéfinir les types
df["Surv"]=pd.Categorical(df["Surv"],ordered=False)
df["Classe"]=pd.Categorical(df["Classe"],
    ordered=False)
df["Genre"]=pd.Categorical(df["Genre"],
    ordered=False)
df["Port"]=pd.Categorical(df["Port"],ordered=False)
df.dtypes
```

Vérifier que les données contiennent des valeurs manquantes, faire des imputations à la médiane d'une valeur quantitative manquante ou la modalité la plus fréquente d'une valeur qualitative absente.

```
df.count()
# imputation des valeurs manquantes
df["Age"]=df["Age"].fillna(df["Age"].median())
df.Port=df["Port"].fillna("S")
```

Continuer en transformant les variables.

```
# Discrétiser les variables quantitatives
df["AgeQ"]=pd.qcut(df_train.Age,3,labels=["Ag1",
   "Ag2","Ag3"])
df["PrixQ"]=pd.qcut(df_train.Prix,3,labels=["Pr1",
   "Pr2","Pr3"])
# redéfinir les noms des modalités
df["Surv"]=df["Surv"].cat.rename_categories(
    ["Vnon","Voui"])
df["Classe"]=df["Classe"].cat.rename_categories(
    ["Cl1","Cl2","Cl3"])
df["Genre"]=df["Genre"].cat.rename_categories(
    ["Gfem","Gmas"])
df["Port"]=df["Port"].cat.rename_categories(
    ["Pc","Pq","Ps"])
df.head()
```

2.2 Analyse en composantes principales

La fonction d'analyse en composantes principales (ACP) est surtout adaptée à l'analyse de signaux, de nombreuses options ne sont pas disponibles notamment les graphiques usuels (biplot, cercle des corrélations...). En revanche des résultats sont liés à la version probabiliste de l'ACP sous hypothèse d'une distribution gaussienne multidimensionnelle des données. Attention, l'ACP est

évidemment centrée mais par réduite. L'option n'est pas prévue et les variables doivent être réduites (fonction `sckit.preprocessing.scale`) avant si c'est nécessaire. L'attribut `transform` désigne les composantes principales, sous-entendu : transformation par réduction de la dimension ; `n_components` fixe le nombre de composantes retenues, par défaut toutes ; l'attribut `components_` contient les `n_components` vecteurs propres mais non normalisés, c'est-à-dire de norme carrée la valeur propre associée et donc à utiliser pour représenter les variables.

```
from sklearn.decomposition import PCA
X=digits.data
y=digits.target
target_name=[0,1,2,3,4,5,6,7,8,9]
# définition de la commande
pca = PCA()
# Estimation, calcul des composantes principales
C = pca.fit(X).transform(X)
# Décroissance de la variance expliquée
plot(pca.explained_variance_ratio_)
show()
```

Diagramme boîte des premières composantes principales.

```
boxplot(C[:,0:20])
show()
```

Représentation des caractères dans le premier plan principal.

```
plt.scatter(C[:,0], C[:,1], c=y, label=target_name)
show()
```

Le même graphique avec une légende mais moins de couleurs.

```
# attention aux indentations
plt.figure()
for c, i, target_name in zip("rgbcmykrgb",
    [0,1,2,3,4,5,6,7,8,9], target_name):
       plt.scatter(C[y == i,0], C[y == i,1],
          c=c, label=target_name)
plt.legend()
plt.title("ACP Digits")
show()
```

Graphique en trois dimensions.

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(1, figsize=(8, 6))
ax = Axes3D(fig, elev=-150, azim=110)
ax.scatter(C[:, 0], C[:, 1], C[:, 2], c=y,
           cmap=plt.cm.Paired)
ax.set_title("ACP: trois premieres composantes")
ax.set_xlabel("Comp1")
ax.w_xaxis.set_ticklabels([])
ax.set_ylabel("Comp2")
ax.w_yaxis.set_ticklabels([])
ax.set_zlabel("Comp3")
ax.w_zaxis.set_ticklabels([])
show()
```

D'autres versions d'analyse en composantes principales sont proposées : *kernel PCA, sparse PCA*...

2.3 Classification non supervisée

Exécution de l'algorithme de classification non supervisée (*clustering*) *k*-means dans un cas simple : le nombre des classes : paramètre `n_clusters` (8 par défaut) est *a priori* connu. D'autres paramètres peuvent être précisés ou laissés à leur valeur par défaut. Le nombre `max_iter` d'itérations (300), le nombre `n_init` (10) d'exécutions parmi lesquelles la meilleure en terme de minimisation de la variance intra-classe est retenue, le mode `init` d'initialisation qui peut être `k-means++` (par défaut) pour en principe accélérer la convergence, `random` parmi les données, ou une matrice déterminant les centres initiaux.

L'option `n_jobs` permet d'exécuter les `n_init` en parallèle. Pour la valeur -2, tous les processeurs sauf un sont utilisés.

Les attributs en sortie contiennent les centres : `cluster_centers_`, les numéros de classe de chaque observation : `labels_`.

De nombreux critères à consulter dans la documentation sont proposés pour évaluer la qualité d'une classification non-supervisée.

On se contente des options par défaut dans cette illustration.

```
from sklearn.cluster import KMeans
from sklearn.metrics import confusion_matrix
est=KMeans(n_clusters=10)
est.fit(X)
classe=est.labels_
print(classe)
```

Les vraies classes étant connues, il est facile de construire une matrice de confusion.

```
import pandas as pd
table=pd.crosstab(df["Surv"],df["Classe"])
print(table)
matshow(table)
title("Matrice de Confusion")
colorbar()
show()
```

Il n'est pas plus difficile de représenter les classes dans les coordonnées de l'ACP. C'est la variable `classe` qui définit les couleurs.

```
fig = plt.figure(1, figsize=(8, 6))
ax = Axes3D(fig, elev=-150, azim=110)
ax.scatter(C[:, 0], C[:, 1], C[:, 2], c=classe,
           cmap=plt.cm.Paired)
ax.set_title("ACP: trois premieres composantes")
ax.set_xlabel("Comp1")
ax.w_xaxis.set_ticklabels([])
ax.set_ylabel("Comp2")
ax.w_yaxis.set_ticklabels([])
ax.set_zlabel("Comp3")
ax.w_zaxis.set_ticklabels([])
show()
```

2.4 Analyse multiple des correspondances

Les données "Titanic" regroupent des variables qualitatives et quantitatives. Après recodage en classes (discrétisation) (cf. le scénario sur l'utilisation de `pandas`) des variables quantitatives, la table obtenue se prête à une analyse factorielle multiple des correspondances. Cette méthode n'est pas présente dans les librairies courantes de python mais disponible sous la forme d'une fonction `mca.py`. Il est possible d'installer la librairie correspondante ou de simplement charger le seul module `mca.py` dans le répertoire courant. Remarque, il ne serait pas plus compliqué de recalculer directement les composantes de l'AFCM à partir de la SVD du tableau disjonctif complet.

Ce tableau est obtenu en remplaçant chaque variables par les indicatrices, ou *dummy variables*, de ses modalités.

```
# Suppression des variables quantitatives
# pour l'AFCM
df_q=df.drop(["Age","Prix"],axis=1)
df_q.head()
# Indicatrices
dc=pd.DataFrame(pd.get_dummies(df_q[["Surv",
  "Classe","Genre","Port","AgeQ","PrixQ"]]))
dc.head()
```

Calcul de l'AFCM et représentations graphiques.

```
import mca
mca_df=mca(dc,benzecri=False)
# Valeurs singulières
print(mca_df.L)
# Composantes principales des colonnes (modalités)
print(mca_df.fs_c())
# Premier plan principal
col=[1,1,2,2,2,3,3,5,5,5,6,6,6,7,7,7]
plt.scatter(mca_df.fs_c()[:, 0],
   mca_df.fs_c()[:, 1],c=col)
for i, j, nom in zip(mca_df.fs_c()[:, 0],
   mca_df.fs_c()[:, 1], dc.columns):
       plt.text(i, j, nom)
```

```
show()
```

Comme pour l'ACP et au contraire de R (cf. FactoMineR), les librairies Python sont pauvres en fonctions graphiques directement adaptées à l'AFCM. Le graphique est construit à partir des fonctions de MatPlotLib. Remarquer l'évidente redondance entre la variable Prix et celle Classe. Il serait opportun d'en déclarer une supplémentaire.

Il est alors facile de construire des classifications non supervisées des modalités des variables ou des passagers à partir de leurs composantes respectives quantitatives, ce qui revient à considérer des distances dites du χ^2 entre ces objets. Très utilisées en marketing (segmentation de clientèle), cette stratégie n'a pas grand intérêt sur ces données.

3 Modélisation et apprentissage statistiques

Voici des exemples de mise en œuvre de quelques unes des méthodes de modélisation / apprentissage statistique les plus utilisées.

3.1 Extraction des échantillons

Le travail préliminaire consiste à séparer les échantillons en une partie *apprentissage* et une autre de *test* pour estimer sans biais l'erreur de prévision. L'optimisation (biais-variance) de la complexité des modèles est réalisée en minimisant l'erreur estimée par validation croisée.

Caractères

```
from sklearn.cross_validation
    import train_test_split
X_train,X_test,y_train,y_test=train_test_split(X,y,
    test_size=0.25,random_state=11)
```

Titanic

Il est nécessaire de transformer les données car scikit-learn ne reconnaît pas la classe DataFrame de pandas ce qui est bien dommage. Les variables qualitatives sont comme précédemment remplacées par les indicatrices de leurs modalités et les variables quantitatives conservées. Cela introduit une évidente redondance dans les données mais les procédures de sélection de modèle feront le tri.

```
# Table de départ
df.head()
# Table des indicatrices
df1=pd.get_dummies(df_q[["Surv","Classe",
    "Genre","Port","AgeQ","PrixQ"]])
# Une seule indicatrice par variable binaire
df1=df1.drop(["Surv_Vnon","Genre_Gmas"],axis=1)
# Variables quantitatives
df2=df[["Age","Prix"]]
# Concaténation
df_c=pd.concat([df1,df2],axis=1)
# Vérification
df_c.columns
```

Extraction des échantillons apprentissage et test.

```
# variables explicatives
T=df_c.drop(["Surv_Voui"],axis=1)
# Variable à modéliser
z=df_c["Surv_Voui"]
# Extractions
T_train,T_test,z_train,z_test=train_test_split(T,z,
    test_size=0.2,random_state=11)
```

Attention : l'échantillon test des données "Titanic" est relativement petit, l'estimation de l'erreur de prévision est donc sujette à caution car probablement de grande variance. Il suffit de changer l'initialisation (paramètre random_state) et ré-exécuter les scripts pour s'en assurer.

3.2 K plus proches voisins

Les images des caractères sont codées par des variables quantitatives. Le problème de reconnaissance de forme ou de discrimination est adapté à l'algorithme des k-plus proches voisins. Le paramètre à optimiser pour contrôler la complexité du modèle est le nombre de voisin n_neighbors. Les options sont décrites dans la documentation.

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=10)
digit_knn=knn.fit(X_train, y_train)
# Estimation de l'erreur de prévision
# sur l'échantillon test
1-digit_knn.score(X_test,y_test)
```

Optimisation du paramètre de complexité du modèle par validation croisée en cherchant l'erreur minimale sur une grille de valeurs du paramètre avec `cv=5`-*fold cross validation* et `n_jobs=1` pour une exécution en parallèle utilisant tous les processeurs sauf 1. Attention, comme la validation croisée est aléatoire, deux exécutions successives ne donnent pas nécessairement le même résultat.

```
from sklearn.grid_search import GridSearchCV
# grille de valeurs
param=[{"n_neighbors":list(range(1,15))}]
knn= GridSearchCV(KNeighborsClassifier(),
   param,cv=5,n_jobs=-1)
digit_knn=knn.fit(X_train, y_train)
# paramètre optimal
digit_knn.best_params_["n_neighbors"]
```

Le modèle est estimé avec la valeur "optimale" du paramètre.

```
knn = KNeighborsClassifier(n_neighbors=
    digit_knn.best_params_["n_neighbors"])
digit_knn=knn.fit(X_train, y_train)
# Estimation de l'erreur de prévision
1-digit_knn.score(X_test,y_test)
# Prévision
y_chap = digit_knn.predict(X_test)
# matrice de confusion
table=pd.crosstab(y_test,y_chap)
print(table)
matshow(table)
title("Matrice de Confusion")
colorbar()
show()
```

3.3 Régression logistique

La prévision de la survie, variable binaire des données "Titanic", se prêtent à une régression logistique. Les versions pénalisées (Ridge, Lasso, elastic net, lars) du modèle linéaire général sont les algorithmes les plus développés dans `scikit-learn` au détriment de ceux plus classique de sélection de variables. Une version Lasso de la régression logistique est testée afin d'introduire la sélection automatique des variables.

Estimation et erreur de prévision du modèle complet sur l'échantillon test.

```
from sklearn.linear_model import LogisticRegression
logit = LogisticRegression()
titan_logit=logit.fit(T_train, z_train)
# Erreur
titan_logit.score(T_test, z_test)
# Coefficients
titan_logit.coef_
```

Comme pour le modèle linéaire, il faudrait construire les commandes d'aide à l'interprétation des résultats.

Pénalisation et optimisation du paramètre par validation croisée. Il existe une fonction spécifique mais son mode d'emploi est peu documenté ; celle `GridSearchCV`, qui vient d'être utilisée, lui est préférée.

```
# grille de valeurs
param=[{"C":[0.01,0.05,0.1,0.15,1,10]}]
logit = GridSearchCV(LogisticRegression(penalty="l1"),
   param,cv=5,n_jobs=-1)
titan_logit=logit.fit(T_train, z_train)
# paramètre optimal
titan_logit.best_params_["C"]
```

Estimation du modèle "optimal".

```
logit = LogisticRegression(C=0.98,penalty="l1")
titan_logit=logit.fit(T_train, z_train)
```

```
# Erreur
titan_logit.score(T_test, z_test)
# Coefficients
titan_logit.coef_
```

3.4 Arbre de décision

Implémentation

Les arbres binaires de décision (CART : *classification and regression trees*) s'appliquent à tous types de variables. Les options de l'algorithme sont décrites dans la documentation. La complexité du modèle est gérée par deux paramètres : `max_depth`, qui détermine le nombre max de feuilles dans l'arbre, et le nombre minimales `min_samples_split` d'observations requises pour rechercher une dichotomie.

Attention : Même s'il s'agit d'une implémentation proche de celle originale proposée par Breiman et al. (1984)[2] il n'existe pas comme dans R (package `rpart`) un paramètre de pénalisation de la déviance du modèle par sa complexité (nombre de feuilles) afin de construire une séquence d'arbres emboîtés dans la perspective d'un élagage (*pruning*) optimal par validation croisée. La fonction générique de *k-fold cross validation* `GridSearchCV` est utilisée pour optimisée le paramètre de profondeur mais avec beaucoup de précision dans l'élagage. Car ce dernier élimine tout un niveau et pas les seules feuilles inutiles à la qualité de la prévision.

En revanche, l'implémentation anticipe sur celles des méthodes d'agrégation de modèles en intégrant les paramètres (nombre de variables tirées, importance...) qui leurs sont spécifiques. D'autre part, la représentation graphique d'un arbre n'est pas incluse et nécessite l'implémentation d'un autre logiciel libre : Graphviz.

Tout ceci souligne encore les objectifs de développement de cette librairie : temps de calcul et prévision brute au détriment d'une recherche d'interprétation. Dans certains exemples éventuellement pas trop compliqués, un arbre élagué de façon optimal peut en effet prévoir à peine moins bien (différence non significative) qu'une agrégation de modèles (forêt aléatoire ou *boosting*) et apporter un éclairage nettement plus pertinent qu'un algorithme de type "boîte noire".

Titanic

Estimation de l'arbre complet.

```
from sklearn.tree import DecisionTreeClassifier
tree=DecisionTreeClassifier()
digit_tree=tree.fit(T_train, z_train)
# Estimation de l'erreur de prévision
1-digit_tree.score(T_test,z_test)
```

Optimisation du paramètre de complexité du modèle par validation croisée en cherchant l'erreur minimale sur une grille de valeurs du paramètre avec `cv=5`-*fold cross validation* et `n_jobs=1` pour une exécution en parallèle utilisant tous les processeurs sauf 1. Attention, comme la validation croisée est aléatoire et un arbre un modèle instable, deux exécutions successives ne donnent pas nécessairement le même résultat.

```
from sklearn.grid_search import GridSearchCV
param=[{"max_depth":list(range(2,10))}]
titan_tree= GridSearchCV(DecisionTreeClassifier(),
   param,cv=5,n_jobs=-1)
titan_opt=titan_tree.fit(T_train, z_train)
# paramètre optimal
titan_opt.best_params_
```

La valeur "optimale" du paramètre est encore trop importante. Une valeur plus faible est utilisée.

```
tree=DecisionTreeClassifier(max_depth=3)
titan_tree=tree.fit(T_train, z_train)
# Estimation de l'erreur de prévision
1-titan_tree.score(T_test,z_test)
# Estimation de l'erreur de prévision
# sur l'échantillon test
1-titan_tree.score(T_test,z_test)
```

Noter l'amélioration de l'erreur.

```
# prévision de l'échantillon test
z_chap = titan_tree.predict(T_test)
```

```
# matrice de confusion
table=pd.crosstab(z_test,z_chap)
```

Tracer l'arbre avec le logiciel Graphviz.

```
from sklearn.tree import export_graphviz
from sklearn.externals.six import StringIO
import pydot
dot_data = StringIO()
export_graphviz(titan_tree, out_file=dot_data)
graph=pydot.graph_from_dot_data(dot_data.getvalue())
graph.write_png("titan_tree.png")
```

L'arbre est généré dans un fichier image à visualiser pour se rende compte qu'il est plutôt mal élagué et pas directement interprétable sans les noms en clair des variables et modalités.

Caractères

La même démarche est utilisée pour ces données.

```
# Arbre complet
tree=DecisionTreeClassifier()
digit_tree=tree.fit(X_train, y_train)
# Estimation de l'erreur de prévision
1-digit_tree.score(X_test,y_test)
```

Optimisation de la profondeur par validation croisée

```
from sklearn.grid_search import GridSearchCV
param=[{"max_depth":list(range(5,15))}]
digit_tree= GridSearchCV(DecisionTreeClassifier(),
   param,cv=5,n_jobs=-1)
digit_opt=digit_tree.fit(X_train, y_train)
digit_opt.best_params_
```

Estimation de l'arbre "optimal".

```
tree=DecisionTreeClassifier(max_depth=11)
digit_tree=tree.fit(X_train, y_train)
```

```
# Estimation de l'erreur de prévision
1-digit_tree.score(X_test,y_test)
```

Prévision de l'échantillon test et matrice de confusion.

```
y_chap = digit_tree.predict(X_test)
# matrice de confusion
table=pd.crosstab(y_test,y_chap)
print(table)
matshow(table)
title("Matrice de Confusion")
colorbar()
show()
```

Cet arbre est bien trop complexe, il est inutile de le tracer, le résultat est illisible.

3.5 Forêts aléatoires

L'algorithme d'agrégation de modèles le plus utilisé est celui des forêts aléatoires (random forest) de Breiman (2001)[1] ce qui ne signifie pas qu'il conduit toujours à la meilleure prévision. Voir la documentation pour la signification de tous les paramètres.

Plus que le nombre d'arbres `n_estimators`, le paramètre à optimiser est le nombre de variables tirées aléatoirement pour la recherche de la division optimale d'un nœud : `max_features`. Par défaut, il prend la valeur $p/3$ en régression et $\sqrt{p}$ en discrimination.

Caractères

```
from sklearn.ensemble import RandomForestClassifier
# définition des paramètres
forest = RandomForestClassifier(n_estimators=500,
   criterion='gini', max_depth=None,
   min_samples_split=2, min_samples_leaf=1,
   max_features='auto', max_leaf_nodes=None,
   bootstrap=True, oob_score=True)
# apprentissage
forest = forest.fit(X_train,y_train)
```

```
print(1-forest.oob_score_)
# erreur de prévision sur le test
1-forest.score(X_test,y_test)
```

L'optimisation du paramètre `max_features` peut être réalisée en minimisant l'erreur de prévision *out-of-bag*. Ce n'est pas prévu, il est aussi possible comme précédemment de minimiser l'erreur par validation croisée.

```
from sklearn.grid_search import GridSearchCV
param=[{"max_features":list(range(4,64,4))}]
digit_rf= GridSearchCV(RandomForestClassifier(
   n_estimators=100),param,cv=5,n_jobs=-1)
digit_rf=digit_rf.fit(X_train, y_train)
# paramètre optimal
digit_rf.best_params_
```

Utilisation de la valeur "optimale". Notez que la méthode n'est pas trop sensible à la valeur de ce paramètre.

```
forest = RandomForestClassifier(n_estimators=500,
   criterion='gini', max_depth=None,
   min_samples_split=2, min_samples_leaf=1,
   max_features=8, max_leaf_nodes=None,
   bootstrap=True, oob_score=True)
# apprentissage
forest = forest.fit(X_train,y_train)
print(1-forest.oob_score_)
# erreur de prévision sur le test
1-forest.score(X_test,y_test)
# prévision
y_chap = forest.predict(X_test)
# matrice de confusion
table=pd.crosstab(y_test,y_chap)
print(table)
matshow(table)
title("Matrice de Confusion")
colorbar()
show()
```

Titanic

Même démarche.

```
# définition des paramètres
forest = RandomForestClassifier(n_estimators=500,
   criterion='gini', max_depth=None,
   min_samples_split=2, min_samples_leaf=1,
   max_features='auto', max_leaf_nodes=None,
   bootstrap=True, oob_score=True)
# apprentissage
forest = forest.fit(T_train,z_train)
print(1-forest.oob_score_)
# erreur de prévision sur le test
1-forest.score(T_test,z_test)
```

Optimisation du paramètre `max_features` par validation croisée.

```
from sklearn.grid_search import GridSearchCV
param=[{"max_features":list(range(2,15))}]
titan_rf= GridSearchCV(RandomForestClassifier(
   n_estimators=100),param,cv=5,n_jobs=-1)
titan_rf=titan_rf.fit(T_train, z_train)
# paramètre optimal
titan_rf.best_params_
```

Utilisation de la valeur "optimale".

```
forest = RandomForestClassifier(n_estimators=500,
   criterion='gini', max_depth=None,
   min_samples_split=2, min_samples_leaf=1,
   max_features=6, max_leaf_nodes=None,
   bootstrap=True, oob_score=True)
# apprentissage
forest = forest.fit(T_train,z_train)
print(1-forest.oob_score_)
# erreur de prévision sur le test
1-forest.score(T_test,z_test)
# prévision
```

```
z_chap = forest.predict(T_test)
# matrice de confusion
table=pd.crosstab(z_test,z_chap)
print(table)
```

Modifier la valeur du paramètre pour constater sa faible influence sur la qualité plutôt médiocre du résultat.

Attention, comme déjà signalé, l'échantillon test est de relativement faible taille (autour de 180), il serait opportun d'itérer l'extraction aléatoire d'échantillons tests pour tenter de réduire la variance de cette estimation et avoir une idée de sa distribution.

Références

[1] L. Breiman, *Random forests*, Machine Learning **45** (2001), 5–32.

[2] L. Breiman, J. Friedman, R. Olshen et C. Stone, *Classification and regression trees*, Wadsworth & Brooks, 1984.

[3] M. Lichman, *UCI Machine Learning Repository*, 2013, http://archive.ics.uci.edu/ml.